Exercise Session

Week 12

Adel Gavranović
agavranovic@student.ethz.ch

Overview

▸ polybox for session material ▸ mail to TA

Today's Topics

Introduction

Iterators

`llvec::init`

Missing Knowledge

Pointers

Dynamic Data Types

Misc

Introduction

- Make sure you have the right handout (Handout12_reupload.pdf). The first Handout has missing slides
- All of the current tasks are running the newest version of the autograder, so if you find any bugs (or typos) send us an email!
- Same goes for the the current Bonus Exercise
- For current Bonus Exercise: max size of board will be set to 16 by 16 (even in the hidden test), so the *efficiency* of your solution is irrelevant
- If you have any questions you can send me an e-mail at *any time* and *any day*. Depending on the problem/question it might take a while to give a good answer (this will still apply even during the Lernphase, so make good use of it)

Follow-up

- Try the ”Push Back” code example again, if you manage that one, everything else will seem much easier

Comments on last [code]expert Exercises

- Don't forget to `&`-reference and `const` your function parameters properly
- What are invariants?
- `read_input`-function in "Task 2a: Complex Numbers"

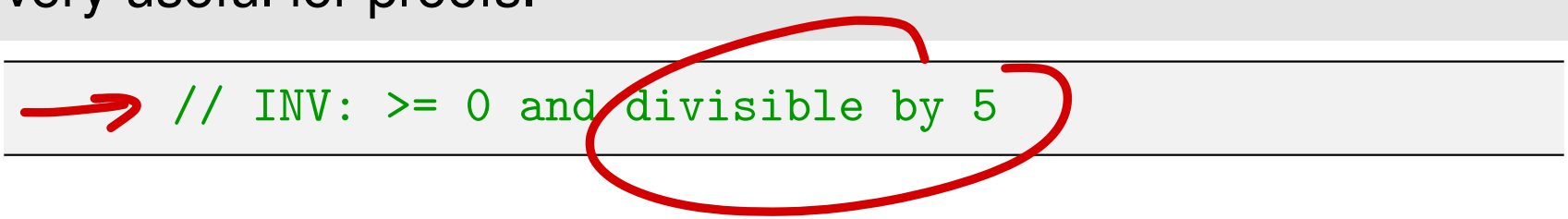
Invariants

Invariants

An invariant is a logical assertion that is always held to be true during a certain phase of execution.

For example, a loop invariant is a condition that is true at the beginning and the end of every execution of a loop.

Invariants *are* allowed to vary, but only *during* a process and then *must be true again at the end of the process*. This can be very useful for proofs.



Task 2a: "Complex Numbers": read_input()

```
// (Slightly altered) STUDENT SOLUTION
bool read_input(std::istream &in, Complex &a){
   bool valid = true;
   char bracket1, comma, bracket2;

   in >> bracket1 >> a.real >> comma >> a.imaginary
      >> bracket2;

   if(bracket1 != '['){
      valid = false;}
   if(comma != ','){
      valid = false;}
   if(bracket2 != ']'){
      valid = false;}

   return valid;
}
```

Credit: L.Z.

Task 2a: "Complex Numbers": read_input()

```
// (Strongly altered) STUDENT SOLUTION
bool read_input(std::istream &in, Complex &a){
    char bracket1, comma, bracket2;

    in >> bracket1 >> a.real >> comma >> a.imaginary
       >> bracket2;

    if(bracket1 != '[' &&
       comma != ',' &&
       bracket2 != ']' && ){
            return true;
    } else {
            return false;
    }
}
```

same

Task 2a: "Complex Numbers": read_input()

```
// MASTER SOLUTION
bool read_input(std::istream &in, Complex &a){
   unsigned char c;
      if(   !(in >> c) || c != '['
         || !(in >> a.real)
         || !(in >> c) || c != ','
         || !(in >> a.imag)
         || !(in >> c) || c != ']' ){
            return false;}
         else{
            return true;
         }
}
```

1. save it in var c
2. return 0: ^:

(in>>c)

0:

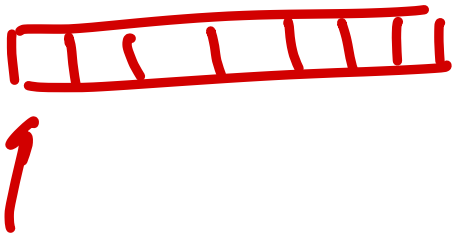
Questions or Comments re: Exercises?

Learning Objectives Checklist

Now I...

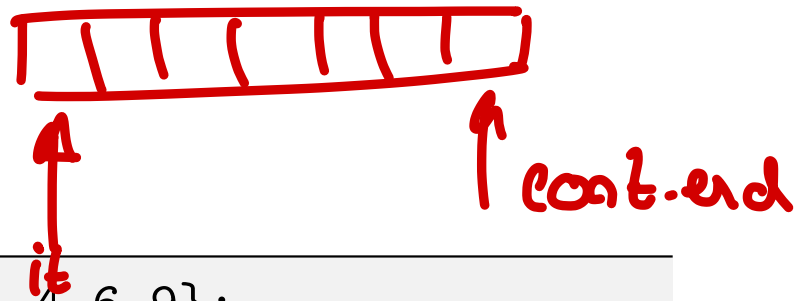
- ☐ can use Iterators for different kinds of containers
- ☐ can implement a simple container

Iterators



- Iterators are a handy way of going through (= iterating over) a `std::set`, `std::vector`, or any other kind of container
- They work very similarly to pointers, so you can use code like `++it` to ”move them forward” and `*it` to access the underlying data
- It helps to think of them as *fancy pointers*

Iterators in Code (std::vector)




```
std::vector<int> cont = {8,3,1,4,6,9};

for (std::vector<int>::iterator it = cont.begin();
     it != cont.end();
     ++it) {

    std::cout << *it << " ";
}
```

Iterators in Code (std::set)

```
std::set<int> cont = {8,3,1,4,6,9};

for (std::vector<int>::iterator it = cont.begin();
     it != cont.end();
     ++it) {

    std::cout << *it << " ";
}
```

Exercise "llvec::init"

Description

The files `vector_linkedlist.h` and `vector_linkedlist.cpp` contain a simplified version of the llvec-vector from the lecture slides. Implement the constructor that initializes the vector with all elements from the iterator.

Hints:
How can you add the first element from the iterator?
How can you add any other element from the iterator?

Personal Hint

Don't "waste" too much time trying to figure out how exactly the other member functions have been implemented. Just read the PRE/POSTs and comments and don't get confused trying to decipher the actual implementations.

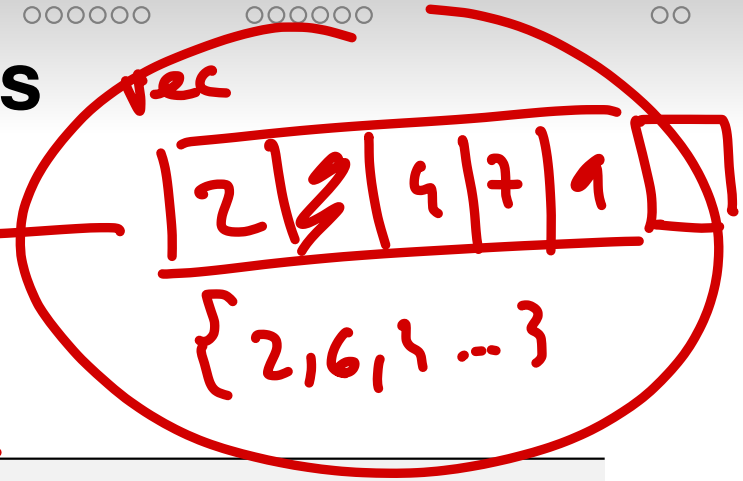
Exercise "`llvec::init`"

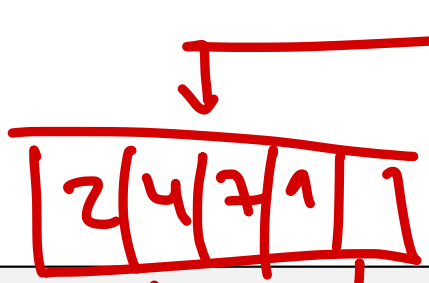
Simpler Description

Implement the constructor `llvec::llvec(begin, end)`

This constructor initializes a new `llvec` and inserts the values that are in a different `llvec` between `begin` and `end`.

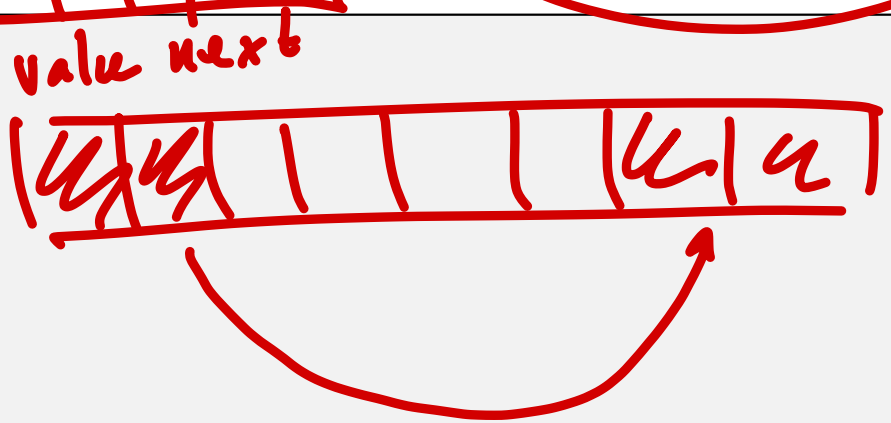
Inside the `llvec`-class: Basics

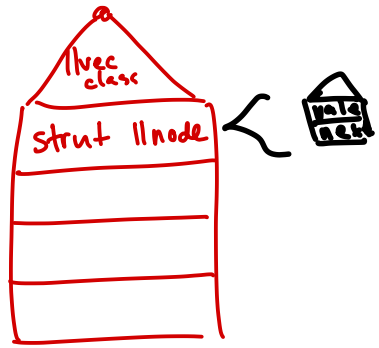





```
struct llnode {
  int value;
  llnode* next;
};

llnode* head;
```





a.begin() —

Inside the `llvec`-class: `const_iterator` class

```
class const_iterator {
    const llnode* node;
public:
    const_iterator(const llnode* const n);
    // PRE: Iterator does not point to the element
    //      beyond the last one.
    // POST: Iterator points to the next element.
    const_iterator& operator++(); // Pre-increment
    // POST: Return the reference to the number at
    //       which the iterator is currently pointing.
    const int& operator*() const;
    // True if iterators are pointing to different
    //      elements.
    bool operator!=(const const_iterator& other) const;
    // True if iterators are pointing to the same
    //      element.
    bool operator==(const const_iterator& other) const;
};
```

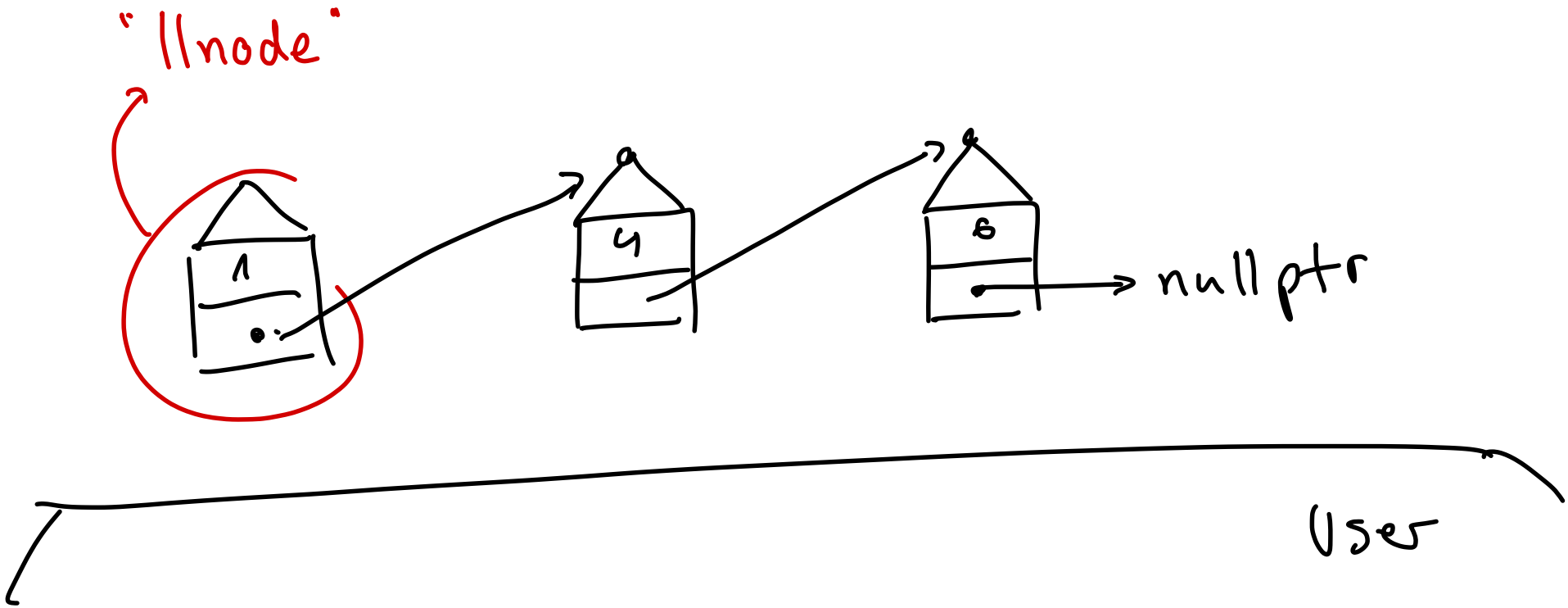
Inside the `llvec`-class: Member Functions

```
// Default Constructor
llvec();

// PRE: begin and end are iterators pointing to the
    same vector and begin is before end.
// POST: The constructed llvec contains all elements
    between begin and end.
llvec(const_iterator begin, const_iterator end);

// POST: e is prepended to the vector.
void push_front(int e);
// POST: Returns an iterator that points to the first
    element.
const_iterator begin() const;
// POST: Returns an iterator that points after the
    last element.
const_iterator end() const;
```

Visualization of `llvec`



1, 4, 6

Conceptual Solution for "`llvec::init`"

"llvec::init" Solution

```
llvec::llvec(llvec::const_iterator begin,
    llvec::const_iterator end) {
  this->head = nullptr;
  if (begin == end) {
    return;
  }

  llvec::const_iterator it = begin;
  // Let's add the first element from the iterator.
    this->head = new llnode{*it, nullptr};
    ++it;
    llnode* current_node = this->head;
  // Let's add all the remaining elements.
    while(it != end){
       current_node->next = new llnode{*it, nullptr};
       current_node = current_node->next;
       ++it;
    }
}
```

Exercise "`llvec::init`"

Task

Do this exercise on your own this evening and try to visualize your solution

Introduction

You're not expected to master recursion, pointers and datastructures (*yet*), so don't panic!

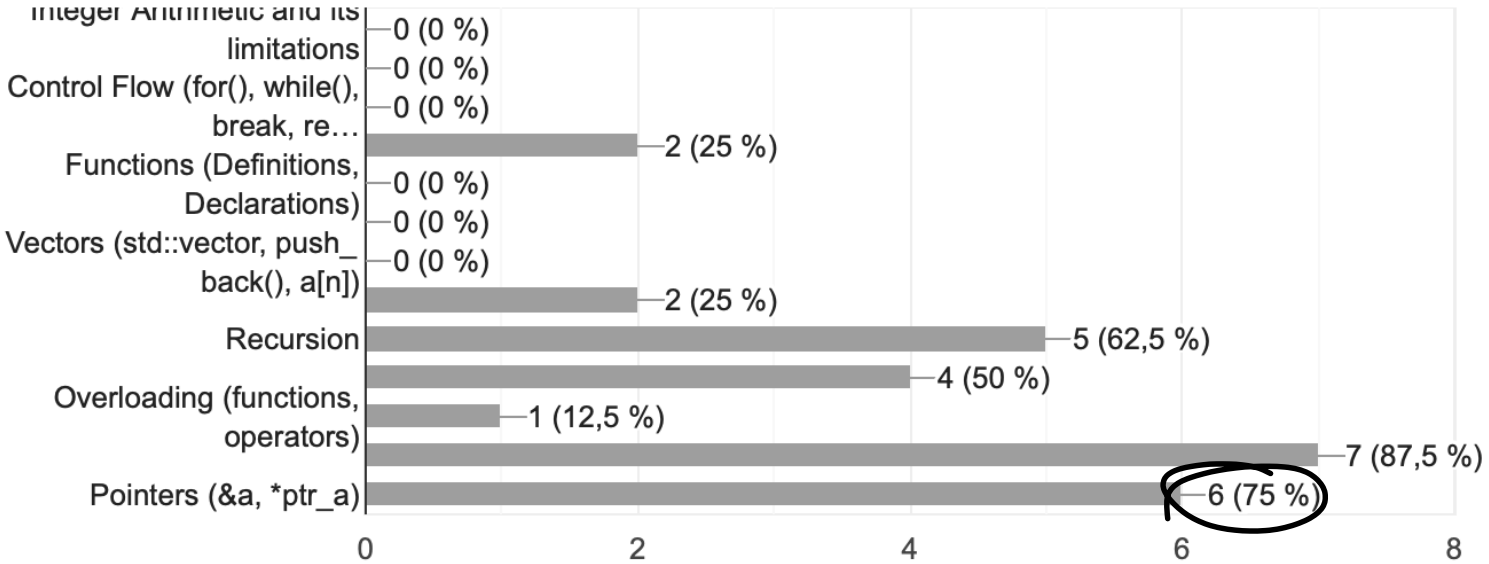
You will understand stuff better the more you (try to) use it.[1]

[1] I know this sounds paradoxical, but there really is no better teacher than good ol' Mr. Practice

Survey Results

Which of these topics (keywords) would you like to revisit in the next exercise session? (Select all that you would like to revisit)

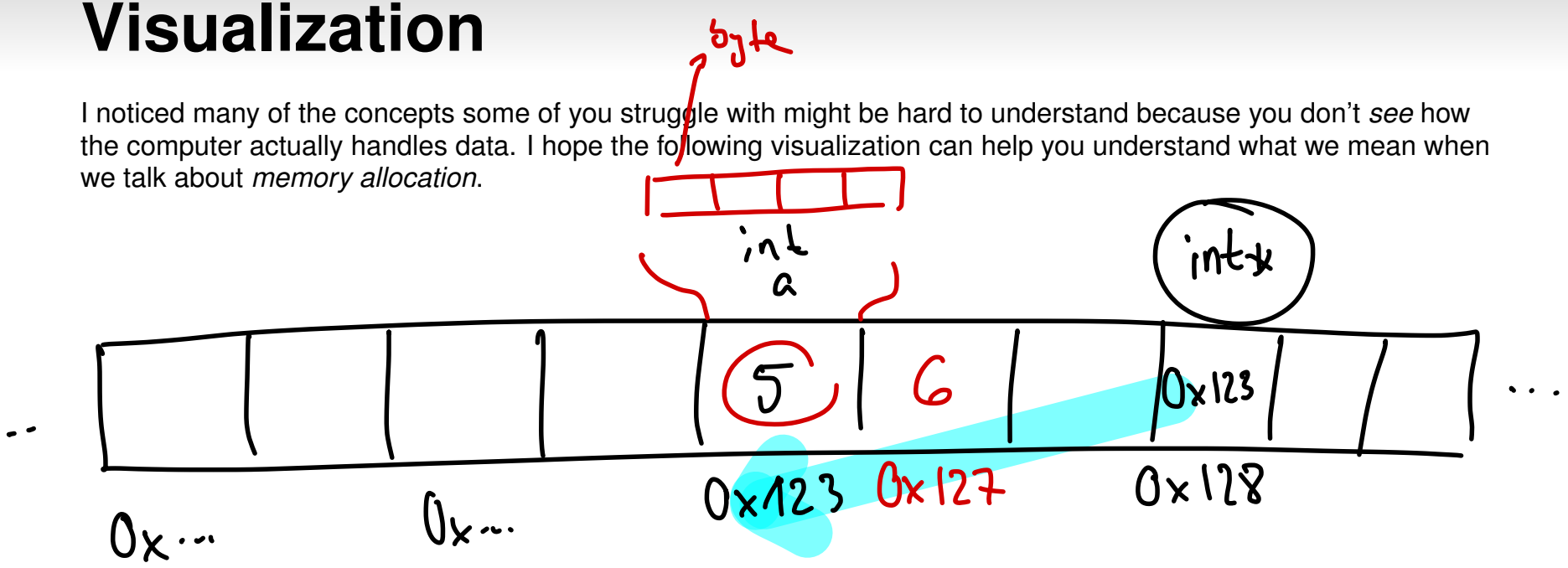
8 Antworten



Thanks to Google froms for the illegible diagramm

Visualization

I noticed many of the concepts some of you struggle with might be hard to understand because you don't *see* how the computer actually handles data. I hope the following visualization can help you understand what we mean when we talk about *memory allocation*.



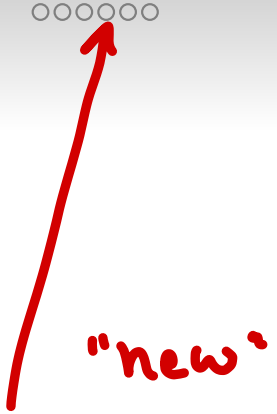
4

ptrs move by the number of bytes that their type dictates, so that it always "jumps" by one value. (++)

```
int* p = &a;
std::cout << *p;
```

dereference operator

Pointers



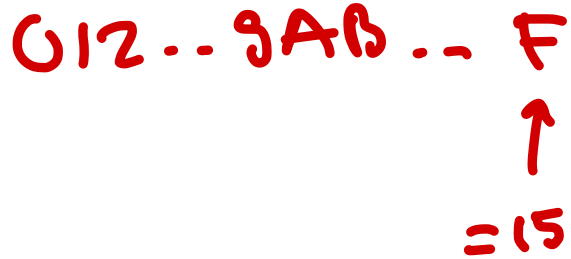
- We use pointers mainly to to keep track of dynamically allocated memory (and to pose complicated exam questions)

Pointers

- We use pointers mainly to to keep track of dynamically allocated memory (and to pose complicated exam questions)
- Seriously, they really are just the memory-address of the thing you let it point to

Pointers

- We use pointers mainly to to keep track of dynamically allocated memory (and to pose complicated exam questions)
- Seriously, they really are just the memory-address of the thing you let it point to
- When `std::cout`-ing them, they usually look something like this: `0xDB11E4` which is just a number in hexidecimal



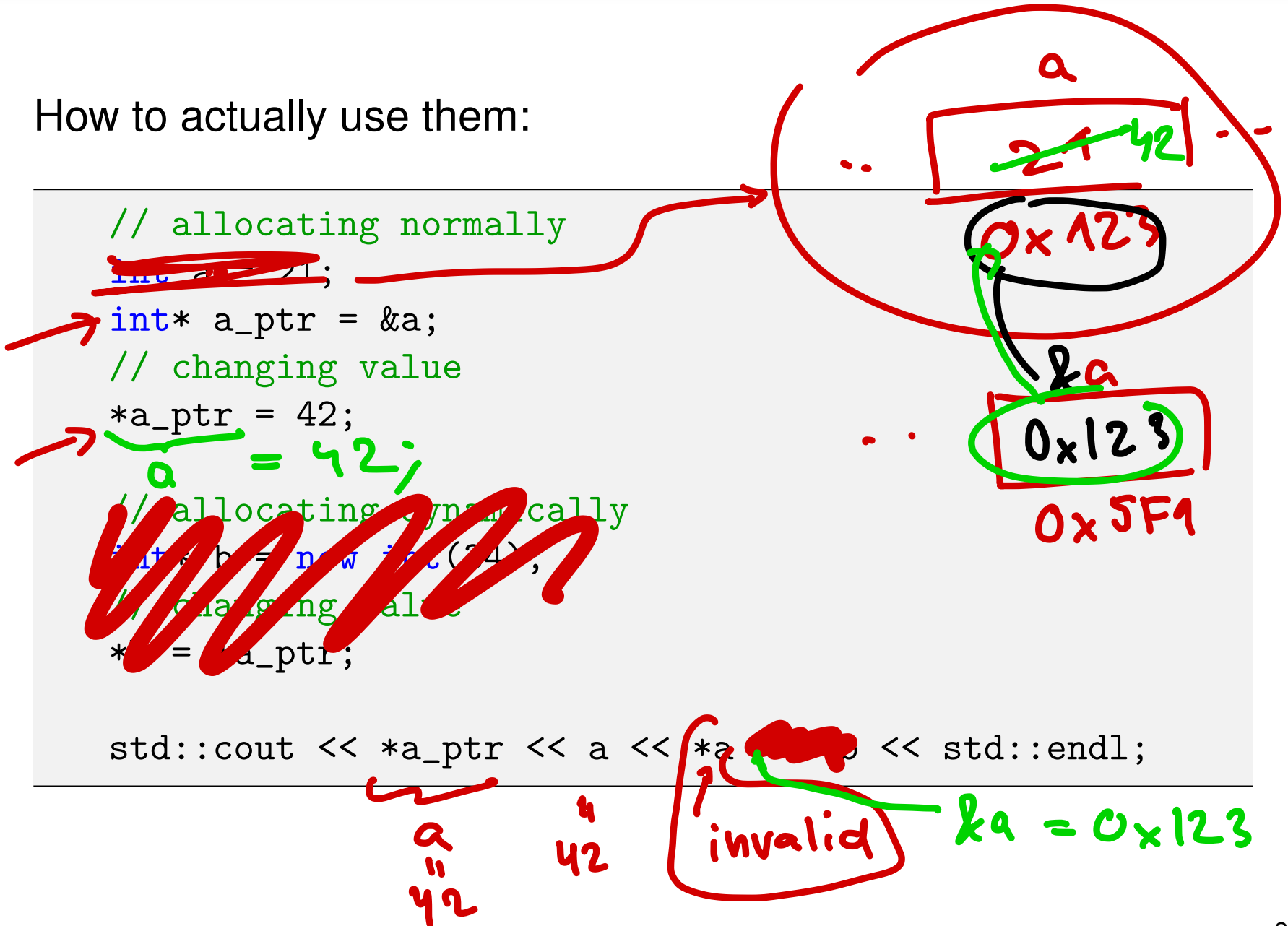
Pointers

How to actually use them:

```
// allocating normally
int a = 21;
int* a_ptr = &a;
// changing value
*a_ptr = 42;

// allocating dynamically
int* b = new int(24);
// changing value
*b = *a_ptr;

std::cout << *a_ptr << a << *b << std::endl;
```



Free Advice

- Make use of the summaries and the internet!

Free Advice

- Make use of the summaries and the internet!
- At the end of the semester (when all summaries are published), save them all into one large PDF and if you stumble upon something you don't understand yet, just search it in this file

Free Advice

- Make use of the summaries and the internet!
- At the end of the semester (when all summaries are published), save them all into one large PDF and if you stumble upon something you don't understand yet, just search it in this file
- Personal recommendation: ▸ the Cherno (C++ videos)

Free Advice

- Make use of the summaries and the internet!
- At the end of the semester (when all summaries are published), save them all into one large PDF and if you stumble upon something you don't understand yet, just search it in this file
- Personal recommendation: ▸ the Cherno
- Practice, Practice, Practice 

Summary on pointers

Zeiger (generell)	**Adresse** eines Objekts im Speicher

Wichtige Befehle:

Definition:	`int* ptr = address_of_type_int;`
(ohne Startwert:	`int* ptr = nullptr;`)
Zugriff auf Zeiger:	`ptr = otr_ptr // Pointer gets new target.`
Zugriff auf Target:	`*ptr = 5 // Target gets new value 5.`
Adresse auslesen:	`int* ptr_to_a = &a; // (a is int-variable)`
Vergleich:	`ptr == otr_ptr // Same target?`
	`ptr != otr_ptr // Different targets?`

(Anstatt `int` gehen natürlich auch andere Typen.)
(Eine `address_of_type_int` kann man durch einen anderen Zeiger oder auch mittels dem Adressoperator `&` erzeugen (siehe Beispiel unten).)

Der **Wert** des Zeigers ist die Speicheradresse des **Targets**. Will man also das Target via diesen Zeiger verändern, muss man zuerst "zu der Adresse gehen". Genau das macht der Dereferenz-Operator `*`.

Beispiel: (Gelte `int a = 5;`)

Wert von `a`:	`5`
Speicheradresse von `a`:	`0x28fef8`
Wert von `a_ptr`:	`0x28fef8`
Wert von `*a_ptr`:	`5`

Ein Zeiger kann immer nur auf den entsprechenden Typ zeigen.
(z.B. `int* ptr = &a;` Hier muss `a` Typ `int` haben.)

Questions?

Dynamic Data Types (new, delete)

- What are they even good for?

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]

std::

int a = 5

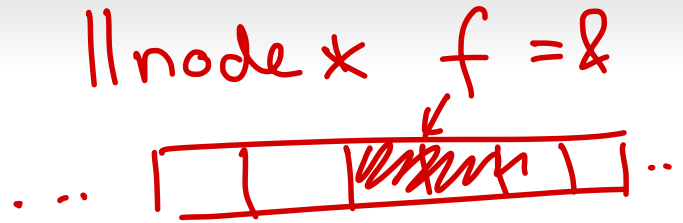
[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types



- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?
  - Whenever we want an object to outlive its scope

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?
  - Whenever we want an object to outlive its scope
  - Whenever the task says we have to

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?
  - Whenever we want an object to outlive its scope
  - Whenever the task says we have to
- How to use them?

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?
  - Whenever we want an object to outlive its scope
  - Whenever the task says we have to
- How to use them?
  - With the keyword `new`

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Data Types

- What are they even good for?
  - Dynamically allocated memory is used when we want to create things that last outside of their initial scope[2]
  - ”Normal” variables get *deconstructed* (= deleted) when the scope in which they were created in ends. A function is also a scope, so every variable that is created inside a function will get deleted if it wasn’t allocated *dynamically*
  - You have to know how to handle pointers in order to use dynamically allocated data
- When to use them?
  - Whenever we want an object to outlive its scope
  - Whenever the task says we have to
- How to use them?
  - With the keyword `new`
  - To each `new` a `delete`! (more on that later)

[2] A scope is usually whatever is inside {swirly brackets}

Dynamic Variables vs ”normal” Variables

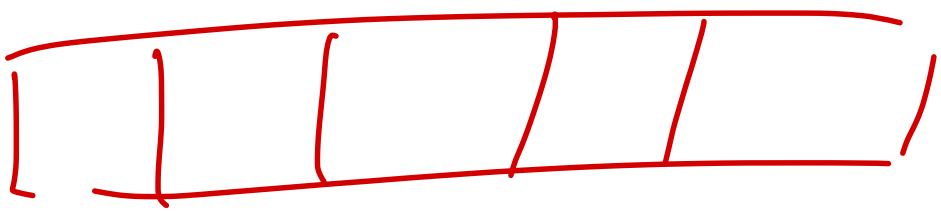
```
// "normally" allocated variable
int n = 42;
// accessing it
n = 1;

// dynamically allocated variable
int* d = new int(42);
// accessing it
*d = 1;
```

char* cp = new char('h');

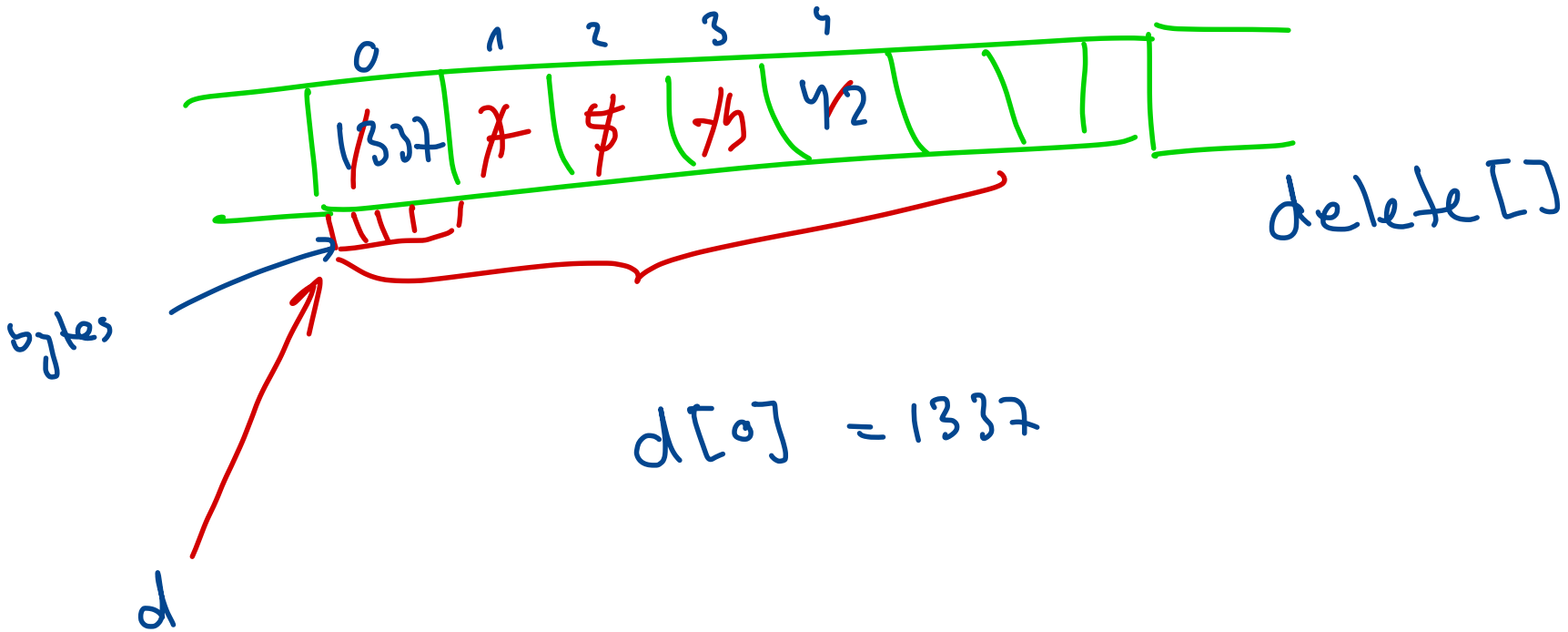
new: "hej, make room for an int"

works if implemented



Dynamic Arrays

```
// dynamically allocated array of variables
int* d = new int[5];
// accessing and modifying
d[0] = 1337;  // first int in array
d[4] = 42;    // last int in array
```



Summary on Dynamic Data Types

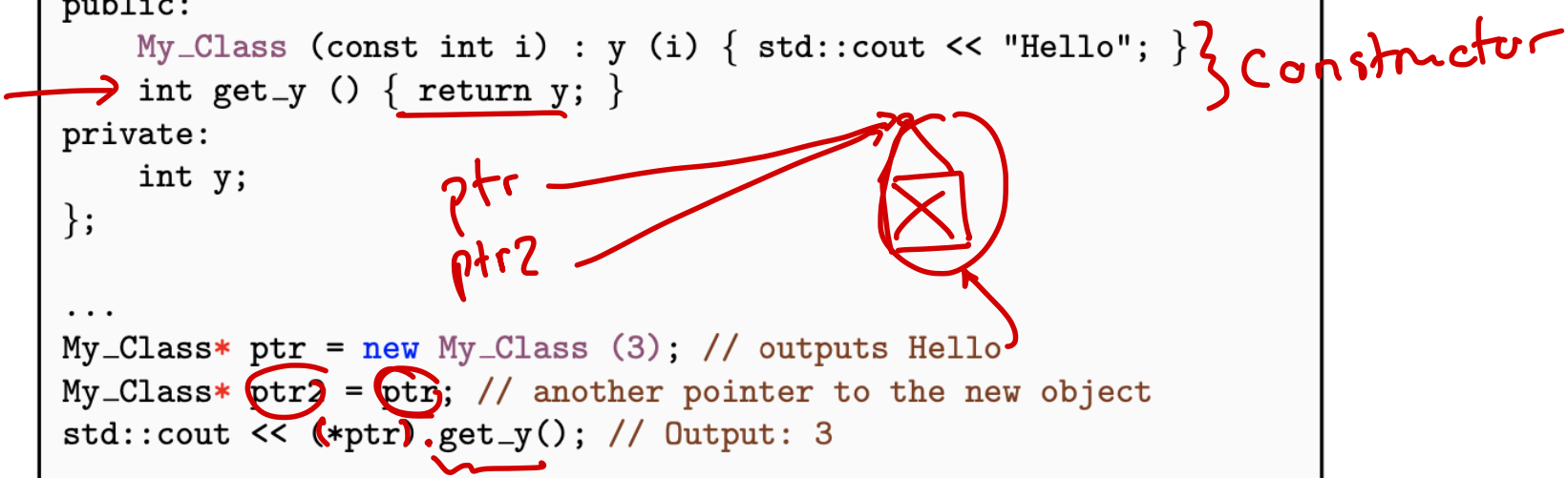
new	**Objekt mit dynamischer Lebensdauer** erstellen.

Mit new wird ein Objekt erstellt, indem der nötige Speicherplatz reserviert wird, und dann ein gegebener Konstruktor aufgerufen wird.

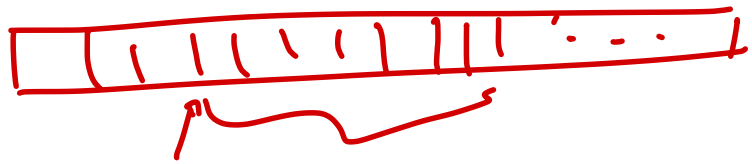
Der Rückgabewert von new ist ein *Pointer* auf das neu erstellte Objekt.

```
Class My_Class {
public:
    My_Class (const int i) : y (i) { std::cout << "Hello"; }
    int get_y () { return y; }
private:
    int y;
};

...
My_Class* ptr = new My_Class (3); // outputs Hello
My_Class* ptr2 = ptr; // another pointer to the new object
std::cout << (*ptr).get_y(); // Output: 3
...
```



Summary on Dynamic Data Types (Arrays)



`new ...[]`	**Ranges mit dynamischer Lebensdauer und Länge** erstellen.

```
int n; std::cin >> n;
int* range = new int[n];
// Read in values to the range
for (int* i = range; i < range + n; ++i) std::cin >> *i;
```

vector v → std::vector

v.begin();
v.push_back();

Questions?

Quick Answers

- **L-value vs R-value**
  - *Basically*, if it has an address it's an L-value, otherwise it's an R-value

Quick Answers

- **L-value vs R-value**
  - *Basically*, if it has an address it's an L-value, otherwise it's an R-value
  - In general, rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables

Quick Answers

- **L-value vs R-value**
  - *Basically*, if it has an address it's an L-value, otherwise it's an R-value
  - In general, rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables
  - Article on that topic ▸ lvalues and rvalues in C++

Quick Answers

- **L-value vs R-value**
  - *Basically*, if it has an address it's an L-value, otherwise it's an R-value
  - In general, rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables
  - Article on that topic ▸ lvalues and rvalues in C++
- **Floats**
  - Please revisit the lecture on that topic and the exercise session notes

Quick Answers

- **L-value vs R-value**
  - *Basically*, if it has an address it's an L-value, otherwise it's an R-value
  - In general, rvalues are temporary and short lived, while lvalues live a longer life since they exist as variables
  - Article on that topic ▸ lvalues and rvalues in C++
- **Floats**
  - Please revisit the lecture on that topic and the exercise session notes
  - Website on that topic ▸ IEEE-754 Floating Point Converter

Post Exercise Session